TP_progDynamiqueCorNB

May 8, 2025

T.P. n°5 Programmation dynamique - corrigé

0.1 1. Rendu de monnaie

On se donne une liste `L` d'entiers ordonnée $a_1, \dots, a_k$ qui représente la valeur faciale de pièces de monnaie. On souhaite déterminer le nombre minimum $r(n,k)$ de pièces parmi $a_1, ..., a_k$ dont la somme vaut n de façon à optimiser le rendu de monnaie.

Par exemple, si $k = 3$ et $L = [1, 2, 5]$ alors : * $r(7, 3) = 2$ (car $7 = 1 \times 5 + 1 \times 2$ et c'est la façon de rendre 7€ qui utilise le moins de pièces). * $r(7, 2) = 4$ (car $7 = 3 \times 2 + 1$).

Remarques :
- On peut utiliser plusieurs fois la même pièce.
- $r(0, k)$ revient à rendre 0€, ce qu'on peut faire avec 0 pièce : $r(0, k) = 0$ - $r(n, 0)$ revient à n'utiliser aucune pièce, ce qui est impossible si $n \neq 0$: on posera $r(n, 0) = \infty$ (`float("inf")` en Python).

> **Question**
> Établir une relation de récurrence vérifiée par $r(n, k)$. On pourra distinguer deux cas pour rendre n euros avec les picèes a_1, ..., a_k selon si a_k est utilisée ou ne l'est pas.
>
> **Réponse**
> Si a_k est utilisée : il faut encore rendre $n - a_k$ euros avec les pièces a_1, ..., a_k (on a le droit d'utiliser plusieurs fois a_k), d'où $r(n, k) = r(n - a_k, k) + 1$.
> Si a_k n'est pas utilisée, $r(n, k) = r(n, k - 1)$.
> Dans le cas général, on considère les deux possibilités et on conserve le minimum :
>
> $$r(n, k) = \min(r(n, k - 1), r(n - a_k, k) + 1)$$
>
> *Remarque : on ne peut utiliser a_k pour rendre n euros que si $n \geq a_k$. Si $n < a_k$, on a donc $r(n, k) = r(n, k - 1)$.*

> **Question**
> En déduire une fonction `rendu(L,n)`, programmée dynamiquement, qui renvoie le nombre minimum de pièces requises pour rendre `n` euros, où `L` est la liste des pièces. Pour ce faire, on définira et remplira la matrice `R` telle que `R[n][k]` contient le nombre minimum de pièces nécessaire pour rendre `n` euros en utilisant les `j` premières pièces de `L`.

[1]:
```
def rendu(L, n):
    k = len(L) # nombre de pièces
    R = [[0]*(k + 1) for _ in range(n + 1)]
    for i in range(1, n + 1):
        R[i][0] = float("inf")
```

```
        for j in range(1, k + 1):
            if i - L[j - 1] >= 0:
                R[i][j] = min(R[i][j - 1], 1 + R[i - L[j - 1]][j])
            else:
                R[i][j] = R[i][j - 1]
    return R[-1][-1]

rendu([1, 2, 5], 7)
```

[1]: 2

Remarque
La mise à jour de la matrice s'effectue dans le sens de lecture classique : dans un ligne de gauche à droite, et de ligne en ligne. La formule de récurrence assure ainsi que seules des valeurs stockées dans des cases précédemment remplies peuvent être appelées.

> *Question bonus*
> Réécrire la fonction précédente par mémoïsation plutôt que par programmation dynamique.

On choisit de créer un dictionnaire qui stocke les valeurs de couples r(n,k) à mesure de leur calcul.

[2]:
```
def rendu_memo(L, n):
    k = len(L)
    d = {}
    def aux(i, j):
        if (i, j) in d:
            return d[(i, j)]
        if i == 0:
            return 0
        if j == 0:
            return float("inf")
        if i - L[j - 1] >= 0:
            d[(i, j)] = min(aux(i, j - 1), 1 + aux(i - L[j - 1], j))
        else:
            d[(i, j)] = aux(i, j - 1)
        return d[(i, j)]
    return aux(n, k)
rendu_memo([1, 2, 5], 7)
```

[2]: 2

0.2 2. Plus grand carré dans une matrice

Étant donnée une matrice carrée remplie de 0 ou 1, on souhaite connaître la taille du plus gros carré de 1 dans cette matrice.
Par exemple, ce nombre est 2 pour la matrice M suivante (correspondant au carré en pointillé) :

La case de coordonnés (x, y) est celle sur la ligne x, colonne y. La case de coordonnées (0, 0) est celle en haut à gauche.

On supposera que les indices en arguments des fonctions ne dépassent pas des tableaux ou matrices correspondants.

Question
Définir `M` en Python comme liste de listes.

[3]:
```
M = [[1, 0, 0, 0], [0, 0, 1, 1], [0, 1, 1, 1], [0, 1, 0, 1]]
```

0.2.1 2.1. Méthode naïve

Question
Écrire une fonction `est_carree(m, x, y, k)` qui détermine si la sous-matrice de `m` de taille $k \times k$ dont la case en haut à gauche a pour coordonnées (`x`, `y`) ne possède que des 1.

[4]:
```
def est_carree(M, x, y, k):
    for i in range(x, x + k):
        for j in range(y, y + k):
            if M[i][j] != 1:
                return False
    return True

assert(est_carree(M, 1, 2, 2) and not est_carree(M, 1, 1, 2))
```

Question
Écrire une fonction `contient_carre` telle que `contient_carre(m, k)` renvoie `True` si `m` contient un carré de 1 de taille k et `False` sinon.

[5]:
```
def contient_carre(M, k):
    n = len(M)
    for i in range(n - k + 1):
        for j in range(n - k + 1):
            if est_carree(M, i, j, k):
                return True
    return False

assert(contient_carre(M, 2) and not contient_carre(M, 3))
```

Question
Écrire une fonction `max_carre1` telle que `max_carre1(m)` renvoie la taille maximum d'un carré de 1 contenu dans `m`.

[6]:
```
def max_carre1(M):
    n = len(M)
    for k in range(n, 0, -1):
        if contient_carre(M, k):
            return k
    return 0
```

```
max_carre1(M)
```

[6]: 2

Question
Quelle est la complexité de `max_carre1(m)` dans le pire cas ?

Réponse
Déterminons la complexité de chacune des fonctions intermédiaires - `est_carre_e(M, x, y, k)` est en $O(k^2)$.
- `contient_carre(M, k)` appelle O(n) fois `est_carre`, donc est en $O(n^2k^2)$.
- `max_carre1(M)` appelle `contient_carre` pour $k = 1, 2, ..., n$, donc est de complexité $\sum_{k=1}^{n} O(n^2k^2) = O(n^2 \sum_{k=1}^{n} k^2)$.
Comme $\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} = O(n^3)$, la complexité totale est $\boxed{O(n^5)}$.

0.2.2 2.2 Méthode améliorée

On va construire une matrice C telle que `C[x][y]` est la taille maximum d'un carré de 1 dans M dont la case en bas à droite est `M[x][y]`.

Par exemple, `C[1][2] = 1` et `C[2][3] = 2` pour la matrice M ci-dessus.

Question
Que valent `C[0][y]` et `C[x][0]` ?

Réponse
`C[0][y] = 0` si `M[0][y] = 0` et `C[0][y] = 1` sinon.
De même pour `C[x][0]`.
Remarque
`C[0][y]` et `C[x][0]` ont ainsi les mêmes valeurs que `M[0][y]` et `M[x][0]`. On peut donc initialiser `C` comme une copie de `M`.

Question
Donner une relation de récurrence vérifiée par `C[x][y]`. On pourra discuter en fonction de la valeur de `M[x][y]`.

Réponse
-> 1er cas : `M[x][y]=0`
Il est alors impossible de trouver un carré de 1 : `C[x][y] = 0`.
-> 2nd cas : `M[x][y]=1`
On pose l'hypothèse de récurrence `C[x][y] = 1 + min(C[x-1][y], C[x][y-1], C[x-1][y-1])`.
Initialisation
Soit il y a un 0 dans les 3 autres du carré, et `C[1][1]=1`, soit il y a 3 1 et `C[1][1]=1+1=2`
Héridité
Supposons la relation vraie et évaluons `C[x+1][y+1]`. Comme précédemment : - si `C[x+1][y]=0` ou `C[x][y+1]=0` ou `C[x][y]=0`, alors `C[x+1][y+1]=1` ce qui convient ; - sinon, notons `m=min(C[x][y+1], C[x+1][y], C[x][y])` et `M[\alpha,\beta]` la case de la matrice de côté de taille m qui contient le 0 associé. On voit que ce 0 appartient nécessairement au carré de côté $m+1$ relatif à `M|x+1,y+1]`, ce qui démontre la relation recherchée.

Question
En déduire une fonction `max_carre2` telle que `max_carre2(M)` renvoie la taille maximum d'un carré de 1 contenu dans `M`, ainsi que les coordonnées de la case en haut à gauche d'un tel carré.

[7]:
```
def maxi(l):
    max_prov=l[0]
    for val in l:
        if val>max_prov:
            max_prov=val
    return max_prov

def max_carre2(m):
    c = m.copy()
    for i in range(len(m)):
        for j in range(len(m[0])):
            if m[i][j] == 1:
                c[i][j] = 1 + min(c[i - 1][j], c[i][j - 1], c[i - 1][j - 1])
    return maxi([maxi(l) for l in c])

max_carre2(M)
```

[7]: 2

Question
Quelle est la complexité de `max_carre2(m)`, en fonction des dimensions de `m`? Comparer avec `max_carre1(m)`.

Réponse
`max_carre2(m)` est en $\boxed{\mathcal{O}(n^2)}$ à cause des deux boucles `for` imbriquées.
C'est donc beaucoup mieux que `max_carre1(m)` qui est en $\mathcal{O}(n^5)$.

0.3 3. Problème du sac à dos

On présente le problème classique dit du sac à dos. Un cambrioleur dispose d'un sac à dos de volume donné qu'il cherche à remplir de façon à maximiser son butin : mieux vaut voler dix bracelets en diamant qu'un seul ordinateur. Pour cela on adopte la modélisation suivante : - en entrée : une capacité c, qui désigne la masse maximale que l'on peut mettre dans le sac, et n objets de masses $w_1, ..., w_n$ et de valeurs v_1, ..., v_n ; - en sortie : la valeur maximum que l'on peut mettre dans le sac.

L'objectif du TP est de comparer différentes approches algorithmiques de ce problème : algorithmes gloutons vs. algorithme dynamique.

0.3.1 3.1 . Algorithmes gloutons

Un algorithme glouton consiste à ajouter des objets un par un au sac, en choisissant à chaque étape l'objet qui a l'air le plus intéressant, si son poids n'excède pas la capacité restante du sac.
Suivant l'ordre dans lequel on choisit les objets, on obtient des algorithmes gloutons différents.

Question

Écrire une fonction `glouton(c, w, v)` qui renvoie la valeur totale des objets choisis par l'algorithme glouton, en considérant les objets dans l'ordre donné par `w` et `v` (on regarde d'abord l'objet de poids `w[0]` et valeur `v[0]`, puis l'objet de poids `w[1]` et valeur `v[1]`…). Tester avec l'exemple ci-dessous. Le résultat est-il optimal ?

[8]:
```
def glouton(c, w, v):
    """Renvoie la valeur maximum qu'on peut obtenir avec les objets
    ordre: liste des objets
    c: capacité du sac
    w: poids des objets
    v: valeur des objets
    """
    poids = 0
    valeur = 0
    for i in range(len(w)):
        if poids + w[i] <= c:
            poids += w[i]
            valeur += v[i]
    return valeur

glouton(10, [5, 3, 6], [4, 4, 6])
```

[8]:
```
8
```

0.3.2 Tri des objets

Question
Écrire une fonction `combine(L1, L2)` qui renvoie la liste des couples `(L1[i], L2[i])`où `L1` et `L2` sont des listes de même longueur.

[9]:
```
def combine(L1, L2):
    L = []
    for i in range(len(L1)):
        L.append((L1[i], L2[i]))
    return L

combine([1, 2, 3], [4, 5, 6])
```

[9]:
```
[(1, 4), (2, 5), (3, 6)]
```

Question
Écrire une fonction `split(c_l)` qui pour une liste de couples `c_l` renvoie deux listes `L1` et `L2` telles que `split(combine(L1,L2))=L1,L2`

[10]:
```
def split(L):
    L1 = []
    L2 = []
    for i in range(len(L)):
        L1.append(L[i][0])
```

```
        L2.append(L[i][1])
    return L1, L2

split([(1, 4), (2, 5), (3, 6)])
```

[10]: ([1, 2, 3], [4, 5, 6])

Si `L` est une liste, `L.sort()` trie `L` par ordre croissant. Si `L` contient des couples, la liste est triée suivant le premier élément de chaque couple (ordre lexicographique).
On peut donner des arguments à la fonction pour : - pour trier par ordre décroissant : `L.sort(reverse=True)` - définir l'ordre de tri selon plusieurs paramètres : `L.sort(key=lambda x: (x[1], x[0]))`

Exemple :

[11]:
```
L = [(1, 4),(1, 1),(9, 12), (7, 5), (3, 6),(9,0)]
L.sort()
print(L)
L.sort(key=lambda x: (x[1], x[0]))
print(L)
```

```
[(1, 1), (1, 4), (3, 6), (7, 5), (9, 0), (9, 12)]
[(9, 0), (1, 1), (1, 4), (7, 5), (3, 6), (9, 12)]
```

Question
Écrire une fonction `tri_poids(w, v)` qui renvoie les listes `w2` et `v2` obtenues à partir de `w` et `v` en triant les poids par ordre croissant. On utilisera `L.sort`, `combine` et `split`.

[12]:
```
def tri_poids(w, v):
    L = combine(w, v)
    L.sort()
    return split(L)

tri_poids([5, 3, 6], [42, 0, 2])
```

[12]: ([3, 5, 6], [0, 42, 2])

0.3.3 Stratégies gloutonnes

Question
En déduire une fonction `glouton_poids(c, w, v)` qui renvoie la valeur totale des objets choisis par l'algorithme glouton, en considérant les objets dans l'ordre de poids croissant. On pourra réutiliser `glouton`.
Cet algorithme est-il toujours optimal ?

[13]:
```
def glouton_poids(c, w, v):
    w, v = tri_poids(w, v)
    return glouton(c, w, v)

glouton_poids(10, [5, 3, 6], [4, 4, 10])
```

[13]: 8

Réponse
Cet algorithme peut ne pas être optimal si des objets très légers sont présents en grand nombre et n'ont qu'une valeur faible.

Question
Écrire de même des fonctions `tri_valeur(w, v)` et `glouton_valeur(c, w, v)` qui renvoie la valeur totale des objets choisis par l'algorithme glouton, en considérant les objets dans l'ordre de valeur décroissante. Cet algorithme est-il toujours optimal ?

[14]:
```
def tri_valeur(w, v):
    L = combine(v, w)
    L.sort(reverse=True)
    L1, L2 = split(L)
    return L2, L1

def glouton_valeur(c, w, v):
    w, v = tri_valeur(w, v)
    return glouton(c, w, v)

glouton_valeur(10, [5, 4, 7], [4, 4, 6])
```

[14]: 6

Réponse
Cet algorithme peut ne pas être optimal si des objets très lourds n'ont que peu de valeur.

Question
De même, écrire une fonction `glouton_ratio(c, w, v)` qui renvoie la valeur totale des objets choisis par l'algorithme glouton, en considérant les objets dans l'ordre de ratio valeur/poids décroissant. On pourra utiliser deux fois `combine`.

[15]:
```
def tri_ratio(v, w):
    L = combine(v, w)
    L = combine([v[i]/w[i] for i in range(len(v))], L)
    L.sort(reverse=True)
    return split(split(L)[1])

def glouton_ratio(c, w, v):
    v, w = tri_ratio(v, w)
    return glouton(c, w, v)

glouton_ratio(10, [5, 4, 7], [4, 4, 6])
```

[15]: 8

0.4 3.2. Programmation dynamique

On considère toujours le même problème avec - en entrée : une capacité c, qui désigne la masse maximale que l'on peut mettre dans le sac, et n objets de masses $w_1, ..., w_n$ et de valeurs v_1, ..., v_n ; - en sortie : la valeur maximum que l'on peut mettre dans le sac.

Soit `dp[c][j]` la valeur maximum que l'on peut mettre dans un sac de capacité c, en ne considérant que les $j \leqslant n$ premiers objets. On suppose que les poids sont strictement positifs.

Question
Que valent `dp[c][0]` et `dp[0][j]` ?

Réponse
`dp[i][0] = 0` : sans objet dans le sac, ou dans un sac de capacité nulle, on ne peut rien emporter.

Question
Exprimer `dp[c][j]` en fonction de `dp[c][j-1]` dans le cas où $w_j > c$.

Réponse
`dp[c][j] = dp[c][j-1]` : on ne peut pas mettre l'objet j dans le sac de capacité c.

Question
Supposons $w_j \leq c$. Donner une formule de récurrence sur `dp[c][j]`, en distinguant le cas où l'objet j est choisi et le cas où il ne l'est pas.

Réponse

$$dp[i][j] = \max(\underbrace{dp[c][j-1]}_{sansprendreo_j}, \underbrace{dp[c-w_j][j-1]+v_j)}_{enprenanto_j, sic-w_j \geq 0}$$

$$

Question
En déduire une fonction `prog_dyn(c, w, v)` qui renvoie la valeur maximum que l'on peut mettre dans un sac de capacité c, en ne considérant que les j premiers objets, en remplissant une matrice `dp` de taille $(c+1) \times (n+1)$.

[16]:
```
def prog_dyn(c, w, v):
    n = len(w)
    dp = [[0 for j in range(c+1)] for i in range(n+1)]
    for i in range(1, n+1):
        for j in range(1, c+1):
            if j < w[i-1]:
                dp[i][j] = dp[i-1][j]
            else:
                dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i-1]] + v[i-1])
    return dp[n][c]

prog_dyn(10, [5, 4, 7], [4, 4, 6])
```

[16]: 8

0.5 3.3. Comparaison des différentes approches

Question
Écrire une fonction `genere_instance()` qui renvoie un triplet (c, w, v), où c est un entier aléatoire entre 1 et 1000 et w, v sont des listes de 100 entiers aléatoires entre 1 et 100.
On importera `random` pour utiliser `random.randint(a, b)` qui génère un entier aléatoire entre a et b inclus.

[17]:
```
import random

def genere_instance():
    c = random.randint(1, 1000)
    w = [random.randint(1, 100) for i in range(100)]
    v = [random.randint(1, 100) for i in range(100)]
    return c, w, v
```

Question
Afficher, pour chaque stratégie gloutonne (ordre de poids, ordre de valeur, ordre de ratio), l'erreur commise par rapport à la solution optimale, en moyennant sur 100 instances générées par `genere_instance()`.
Quelle stratégie gloutonne semble être la plus efficace ?

[18]:
```
gp, gv, gr = 0, 0, 0
for i in range(100):
    c, w, v = genere_instance()
    sol = prog_dyn(c, w, v)
    gp += glouton_poids(c, w, v)/sol
    gv += glouton_valeur(c, w, v)/sol
    gr += glouton_ratio(c, w, v)/sol
print(f"Glouton poids : {gp/100}")
print(f"Glouton valeur : {gv/100}")
print(f"Glouton ratio : {gr/100}")
```

```
Glouton poids : 0.8607063896595035
Glouton valeur : 0.596637180085549
Glouton ratio : 0.9951070140876425
```

Question
Comparer le temps total d'exécution de la stratégie gloutonne par ratio et de la programmation dynamique, sur 100 instances générées par `genere_instance()`. On pourra importer `time` et utiliser `time.time()` pour obtenir le temps actuel en secondes. Conclure

[19]:
```
import time

t1, t2 = 0, 0
for i in range(100):
    c, w, v = genere_instance()
```

```
    t = time.time()
    glouton_poids(c, w, v)
    t1 += time.time() - t
    t = time.time()
    prog_dyn(c, w, v)
    t2 += time.time() - t
print(f"Glouton poids : {t1} s")
print(f"Programmation dynamique : {t2} s")
```

```
Glouton poids : 0.0021364688873291016 s
Programmation dynamique : 0.4671156406402588 s
```

0.6 3.4. Obtention de la liste des objets choisis

Question

Réécrire la fonction `prog_dyn(c, w, v)` pour qu'elle renvoie la liste des objets choisis. Pour cela, on peut construire la matrice `dp` et remarquer que :
- si `dp[c][j] = dp[c][j-1]`, alors l'objet j n'est pas choisi ; - si `dp[c][j] = dp[c - w_j][j - 1] + v_j`, alors l'objet j est choisi.
On peut donc construire la liste des objets choisis en remontant la matrice `dp` à partir de la case (c, n).

[20]:
```
def prog_dyn(c, w, v):
    n = len(w)
    dp = [[0 for j in range(c+1)] for i in range(n+1)]
    for i in range(1, n+1):
        for j in range(1, c+1):
            if j < w[i-1]:
                dp[i][j] = dp[i-1][j]
            else:
                dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i-1]] + v[i-1])

    # reconstruction de la solution
    i, j = n, c
    sol = []
    while i > 0 and j > 0:
        if dp[i][j] == dp[i-1][j]:
            i -= 1
        else:
            sol.append(i-1)
            j -= w[i-1]
            i -= 1
    return sol

prog_dyn(10, [5, 4, 7], [4, 4, 6])
# la solution optimale consiste à choisir les objets 1 et 0
```

[20]:
```
[1, 0]
```

[ ]: